MCMC convergence diagnosis and model checking in a nutshell (and under construction…)

Masco - 2024

Introduction

- We previously described how to build a Bayesian model (likelihood, prior), how to exploit the resulting posterior distribution.
- Posterior distributions are difficult to derive and relies on the *Jags-"black box"*, the MCMC underlying algorithms.
- We expect the **chains** obtained using MCMC algorithms to **converge to a stationary distribution** which coincides with our target distribution, the joint posterior.
- We cannot check if it converged to our target, but we can check if it converges.
- We describe
    - visual inspection: trace plot, mean running plot, autocorrelation.
    - statistical: Gelman and Rubin test, Potential scale reduction factor.
- Remedies:
    - change point initialization, number of MC iterations.
    - Thinning, burning...

An example: Memory retention

```
t     = c(1, 2, 4, 7, 12, 21, 35, 59, 99, 200)
nt    = length(t)
slist = 1:4
ns    = length(slist)
k = matrix(c(18, 18, 16, 13, 9, 6, 4, 4, 4, NA,
             17, 13,  9,  6, 4, 4, 4, 4, 4, NA,
             14, 10,  6,  4, 4, 4, 4, 4, 4, NA,
             NA, NA, NA, NA,NA,NA,NA,NA,NA, NA), nrow=ns, ncol=nt, byr
n = 18
data = list("k", "n", "t", "ns", "nt") # to be passed on to JAGS
myinits =    inits <- list(
  list(alpha = 0.8, beta = 0.1),  # Initial values for chain 1
  list(alpha = 0.6, beta = 0.1), # Initial values for chain 2
  list(alpha = 0.4, beta = 0.1)  # Initial values for chain 3
)
# parameters to be monitored:
parameters = c("alpha", "beta")
```

An example: Memory retention

```
# model
model_code = '
# Retention With No Individual Differences
model{
  # Observed and Predicted Data
  for (i in 1:ns){
    for (j in 1:nt){
      k[i,j] ~ dbin(theta[i,j],n)
      #predk[i,j] ~ dbin(theta[i,j],n)
    }
  }
  # Retention Rate At Each Lag For Each Subject Decays Exponentially
  for (i in 1:ns){
    for (j in 1:nt){
      theta[i,j] <- min(1,exp(-alpha*t[j])+beta)
    }
  }
  # Priors
  alpha ~ dbeta(1,1)
  beta ~ dbeta(1,1)
}
'
```

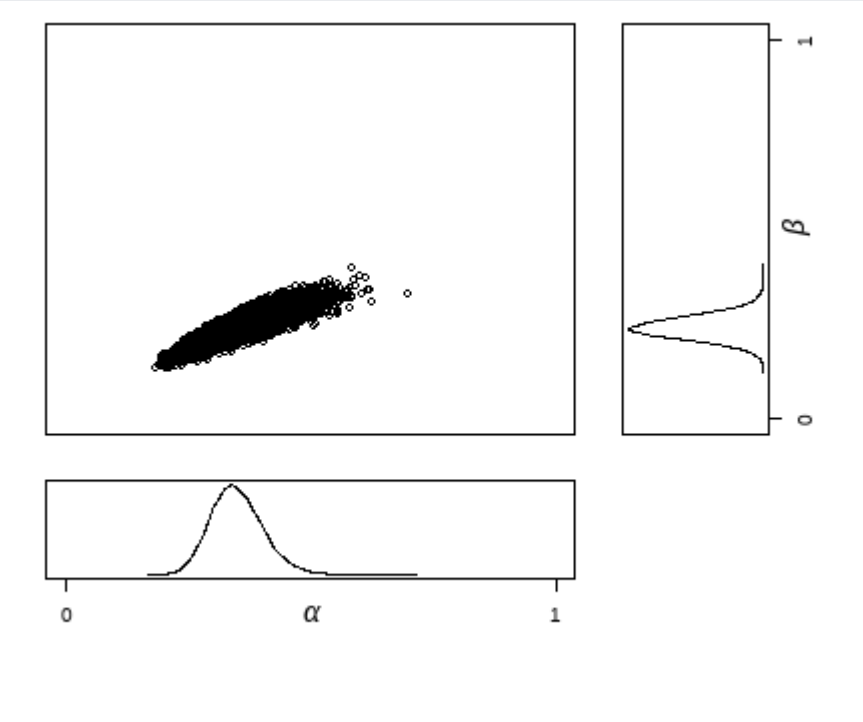
An example: Memory retention

```
library(R2jags)
memory_model = jags(data, inits=myinits, parameters,
                    model.file =textConnection(model_code),
                    n.chains=3, n.iter=10000, n.burnin=1, n.thin=1, DIC=T)
```

```
## module glm loaded

## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 27
##    Unobserved stochastic nodes: 15
##    Total graph size: 97
##
## Initializing model
```

Plots



Posterior summary

Code | Post. summary

We use the package 'coda'

```
library(coda)
```

Transform the 'jags' output as an *mcmc* object.

```
mcmc_model = as.mcmc(memory_model)
```

Posterior summary

```
summary(mcmc_model)
```

Posterior summary

Code | Post. summary

```
##
## Iterations = 2:10000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 9999
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##              Mean      SD  Naive SE Time-series SE
## alpha      0.3488 0.05358 0.0003093      0.0008819
## beta       0.2398 0.03054 0.0001763      0.0004887
## deviance 133.1195 2.05266 0.0118517      0.0242280
##
## 2. Quantiles for each variable:
##
##              2.5%      25%      50%      75%    97.5%
## alpha      0.2537   0.3115   0.3451   0.3821   0.4647
## beta       0.1810   0.2191   0.2390   0.2601   0.3011
## deviance 131.1425 131.6646 132.5003 133.8931 138.6685
```

Effective sample

ESS | ACF in R | ESS in R

- MCMC samples are **not independent**.
- The underlying algorithms generate samples sequentially, correlated samples.
- Autocorrelation reduces the "amount of information"
- Effective Sample Size **ESS** quantifies how many independent samples the correlated MCCM output represents.

$$ESS = \frac{n}{1 + 2\sum_k \rho_k},$$

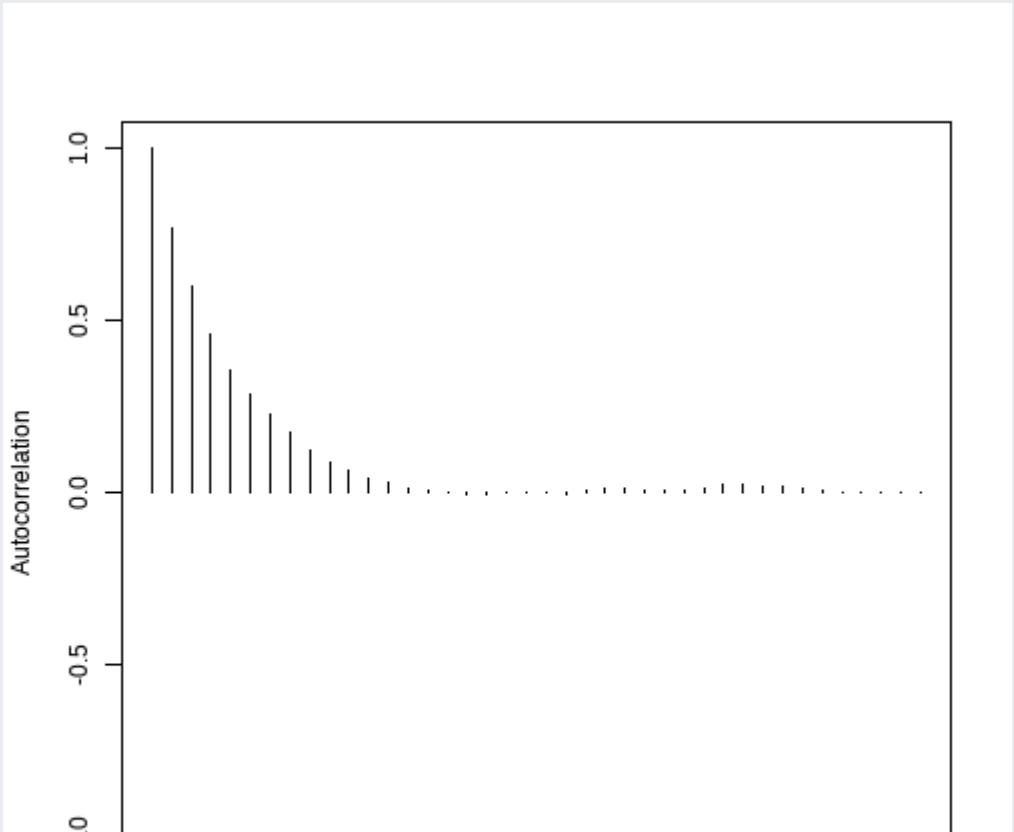
where n is the total number of MCMC iterations, ρ_k is the autocorrelation at lag k.

$$\rho_k = \frac{\sum_{m=1}^{n-k}(\theta_m - \bar{\theta})(\theta_{m+k} - \bar{\theta})}{\sum_{m=1}^{n-k}(\theta_m - \bar{\theta})^2}.$$

Effective sample

ESS | ACF in R | ESS in R

```
autocorr.plot(mcmc_model[,"alpha"])
```



Effective sample

ESS | ACF in R | ESS in R

```
effectiveSize(mcmc_model)
```

```
##    alpha     beta deviance
## 3702.474 3906.504 7398.499
```

Naive Standard Error

The 'naive standard error' of the mean capture the simulation error of the mean, not the posterior uncertainty.

$$NSE = \frac{\text{posterior SD}}{\sqrt{n}}.$$

The time series error adjusts the "naive" standard error for autocorrelation.

DIC

The **Deviance** is a goodness of fit measure of a statistical model.

It is defined as

$$D(\theta) = -2\log(p(y|\theta)) + C$$

, where C is a constant that depends only on the data.

Since C is constant with respect to the model, it is often ignored in model comparisons.

Lower deviance values indicate a better fit of the model to the data

The deviance is automatically computed for every iteration of the Markov Chain Monte Carlo (MCMC) sampling. It is used for:

Deviance Information Criteria (DIC), is defined as

$$DIC = \bar{D} + p_D$$

where,

$\bar{D}$ is the posterior mean of the deviance.

p_D is the "effective number of parameters," which measures model complexity.

The deviance balances model fit and complexity, ensuring a robust model selection process.

Visual inspection

One method to assess whether an MCMC chain has converged is to evaluate how well it is **mixing**, i.e., how effectively it is exploring the parameter space. If the chain moves slowly or gets stuck in certain regions of the parameter space, it indicates poor mixing, which can delay convergence.

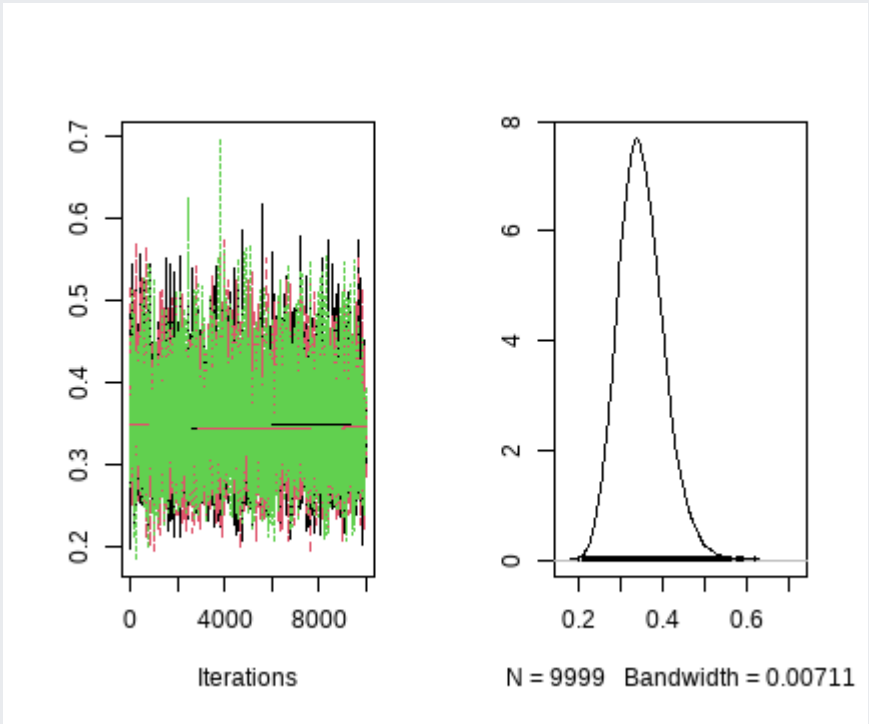
Visual inspection is a useful tool for assessing the mixing behavior of the chain. This involves examining **trace plots for each parameter** to identify patterns or irregularities. Ideally, a well-mixed chain should exhibit a "hairy caterpillar" appearance, with no obvious trends, periodicity, or long periods of stagnation. It is essential to perform these inspections for every parameter in the model to ensure that all components of the parameter space are being adequately explored.

A traceplot is a plot of the iteration number against the value of the draw of the parameter at each iteration. We can see whether our chain gets stuck in certain areas of the parameter space, which indicates bad mixing.

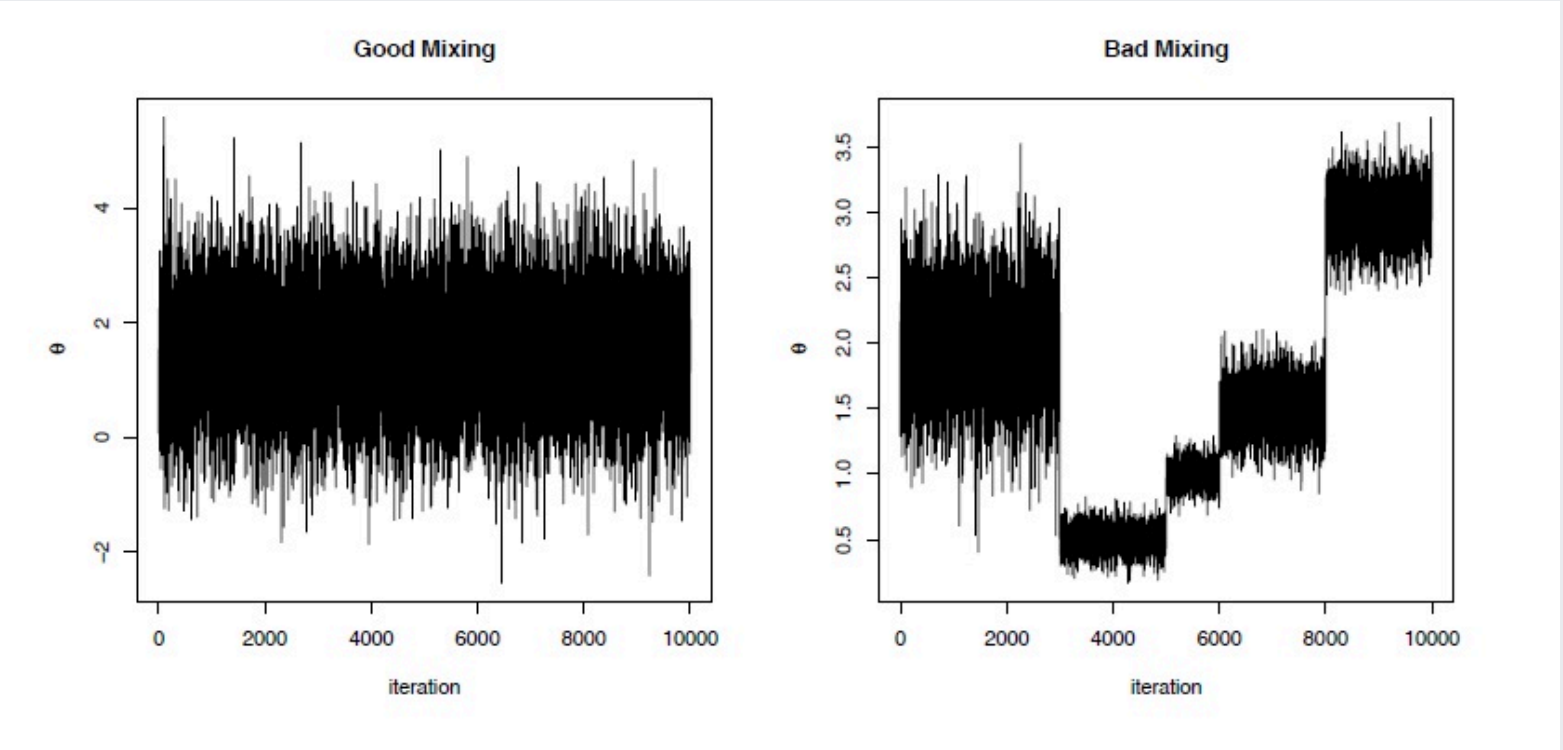
Traceplot for the memory retention example

We can do traceplots and density plots by plotting an mcmc object or by calling the traceplot() and densplot() functions.

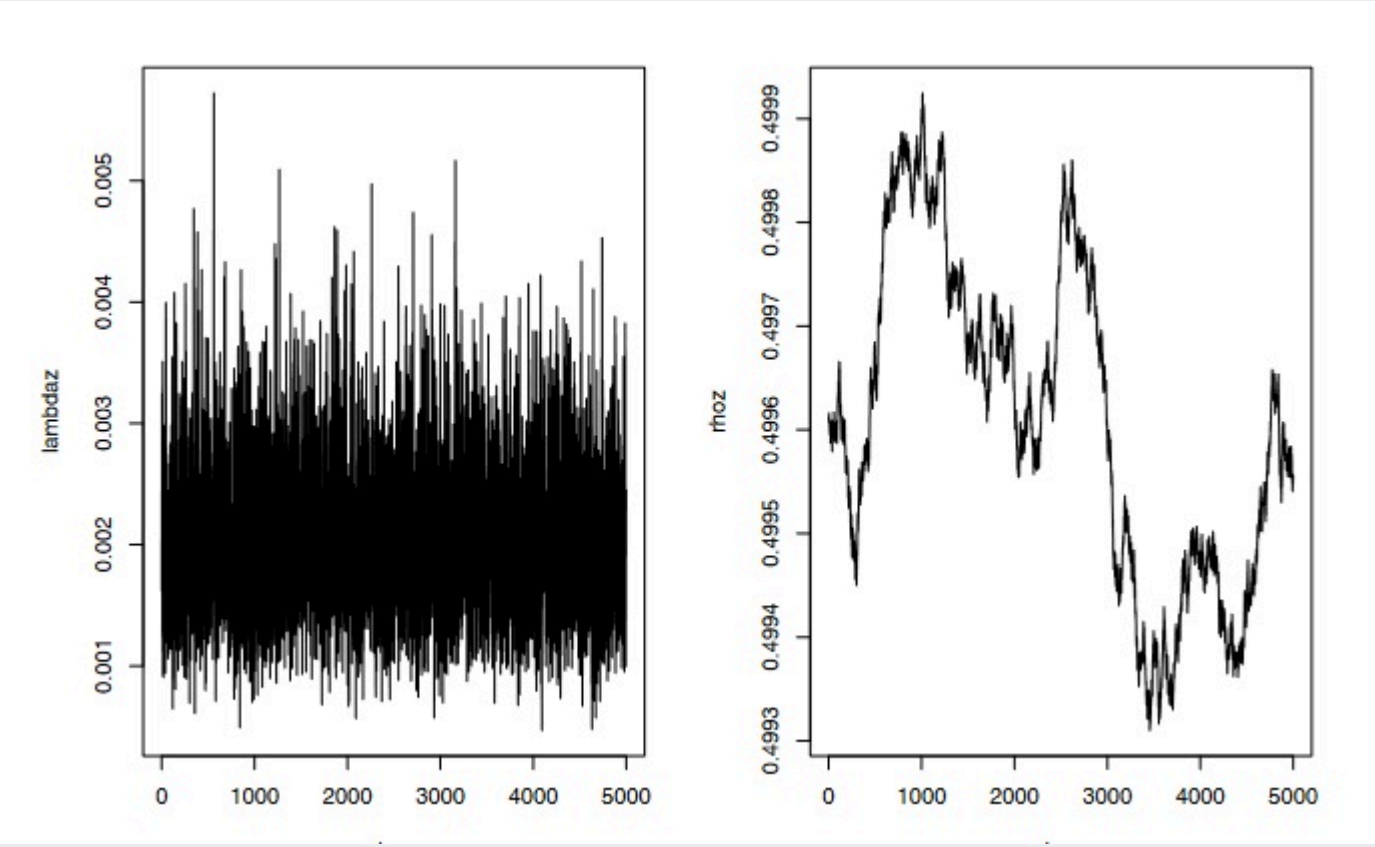
```
plot(mcmc_model[,"alpha"])
```



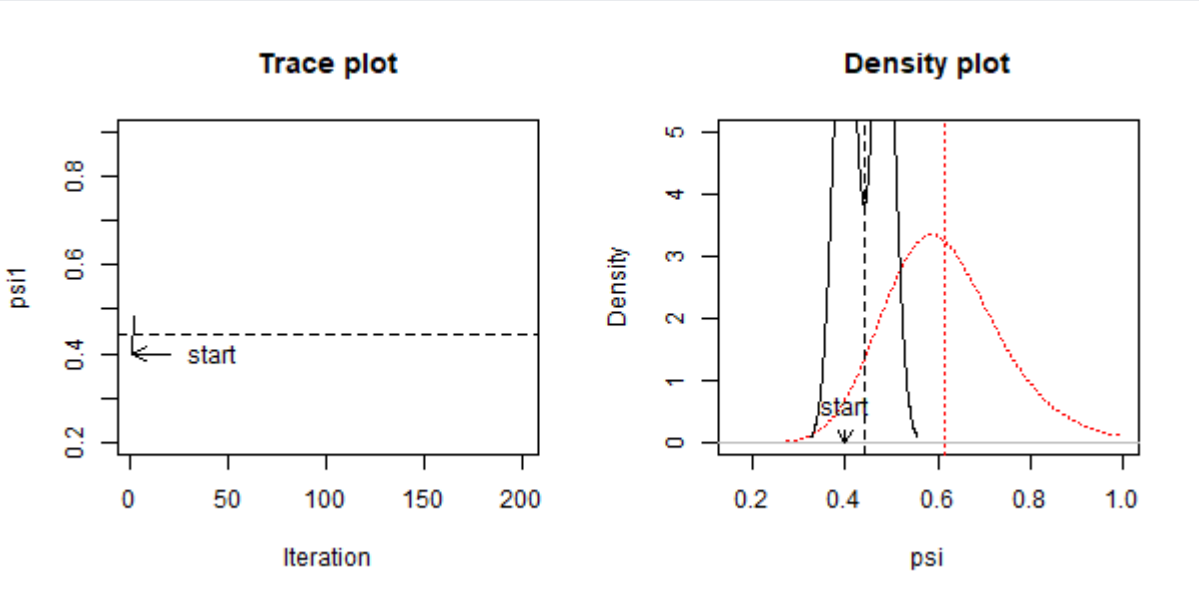
Good and Bad Traceplots



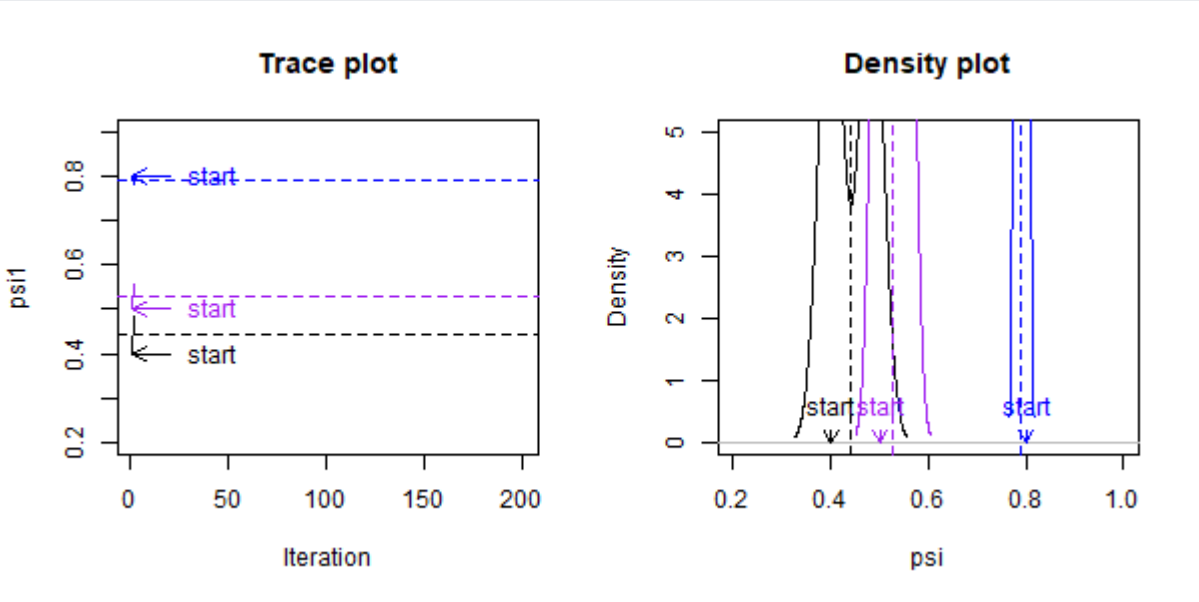
Good and Bad Traceplots



Traceplot and density



Traceplot and density of few chains



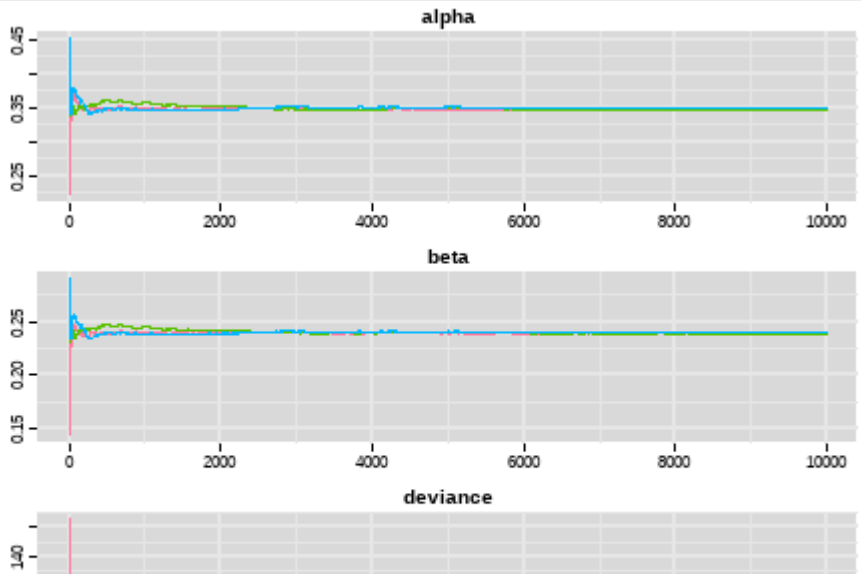
Running Mean plot

We can also use running mean plots to check how well our chains are mixing. A running mean plot is a plot of the iterations against the mean of the draws up to each iteration.

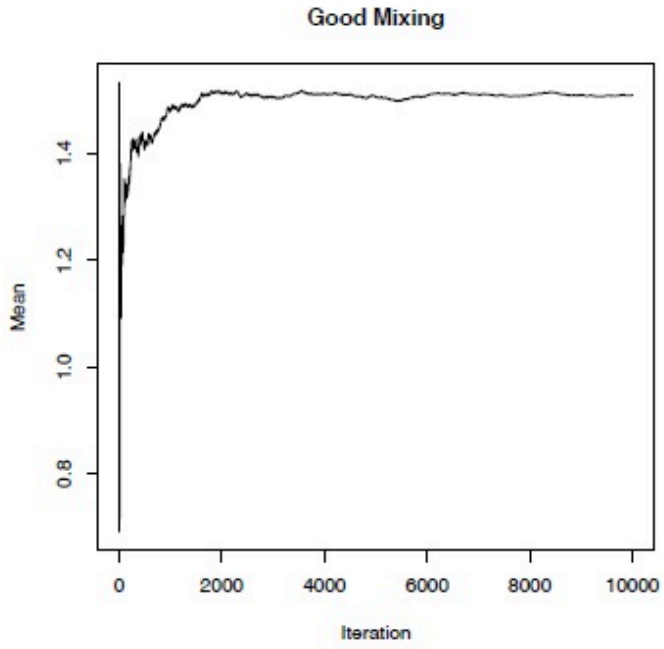
```
library(mcmcplots, quietly = TRUE)
```

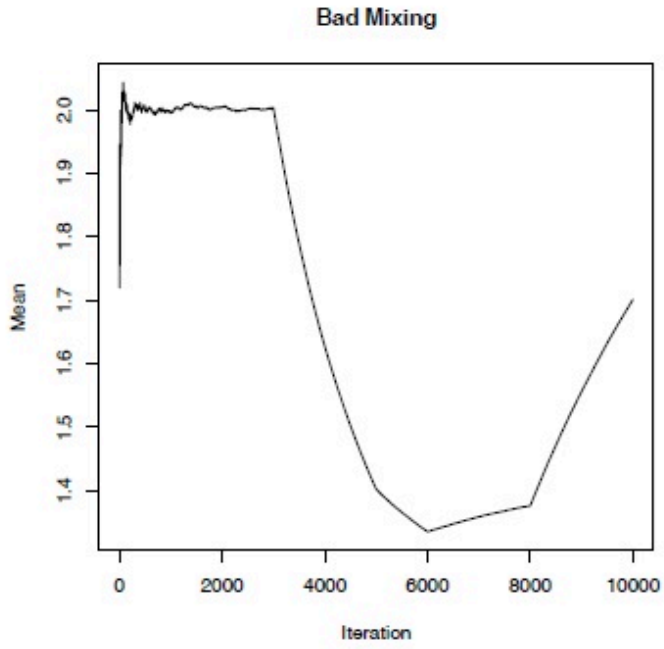
```
## Registered S3 method overwritten by 'mcmcplots':
##   method        from
##   as.mcmc.rjags R2jags
```

```
rmeanplot(mcmc_model)
```

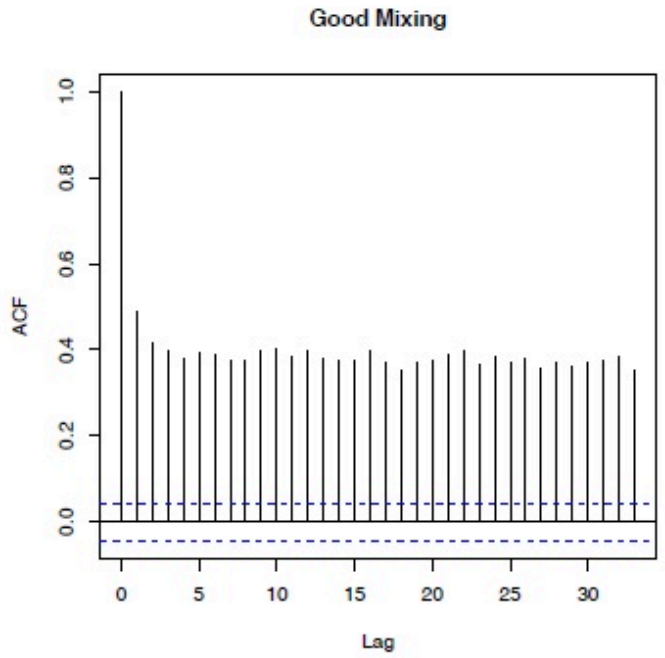


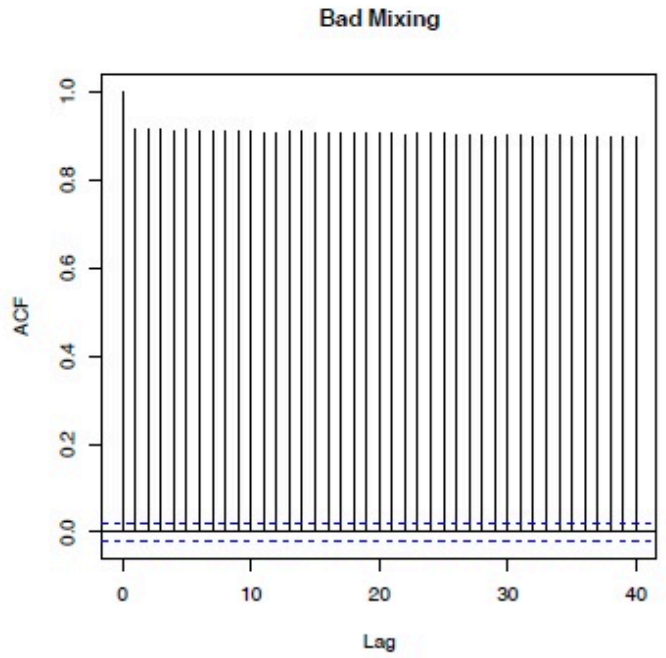
Good and Bad running mean plot





Good and Bad ACF





Gelman and Rubin Multiple Sequence Diagnostic

Steps (for each parameter):

1. Run $m \geq 2$ chains of length $2n$ from overdispersed starting values.
2. Discard the first n draws in each chain.
3. Calculate the within-chain and between-chain variance.
4. Calculate the estimated variance of the parameter as a weighted sum of the within-chain and between-chain variance.
5. Calculate the potential scale reduction factor.

Between-Within Chain Variance

Within chain variance

$$W = \frac{1}{m} \sum_{j=1}^{m} s_j^2,$$

where

$$s_j^2 = \frac{1}{n-1} \sum_{i=1}^{n} (\theta_{ij} - \bar{\theta}_j)^2.$$

s_j is the variance of the j-th chain. W is the mean of the variances of each chain. It is likely to underestimate the true variance of the stationary distribution since our chains have probably not reached all the points of the stationary distribution.

Between chain variance

$$B = \frac{n}{m-1} \sum_{j=1}^{m} (\bar{\theta}_j - \bar{\bar{\theta}})^2.$$

This is the variance of the chain means multiplied by n because each chain is based on n draws.

Estimated Variance

We can then estimate the variance of the stationary distribution as a weighted average of W and B.

$$\widehat{Var}(\theta) = \left(1 - \frac{1}{n}\right) W + \frac{1}{n} B.$$

Because of overdispersion of the starting values, this overestimates the true variance, but it is unbiased if the starting distribution equals the stationary distribution (if starting values were not overdispersed).

Potential Scale Reduction Factor

The potential scale reduction factor is

$$\hat{R} = \sqrt{\frac{\widehat{Var}(\theta)}{W}}.$$

numerator and denominator are both unbiased estimates of the variance if the two chains have converged

otherwise W is an underestimate (hasn't explored enough)

numerator will overestimate as B is too large (overdispersed starting points)

As $n \to \infty$ and $B \to 0$, $R \to 1$.

When $\hat{R}$ is high (greater than 1.1 or 1.2), then we should run our chains out longer to improve convergence to the stationary distribution.

If we have more than one parameter, then we need to calculate the potential scale reduction factor for each parameter.

We should run our chains out long enough so that all the potential scale reduction factors are small enough. We can then combine the mn total draws from our chains to produce one chain from the stationary distribution.

Potential scale reduction factor

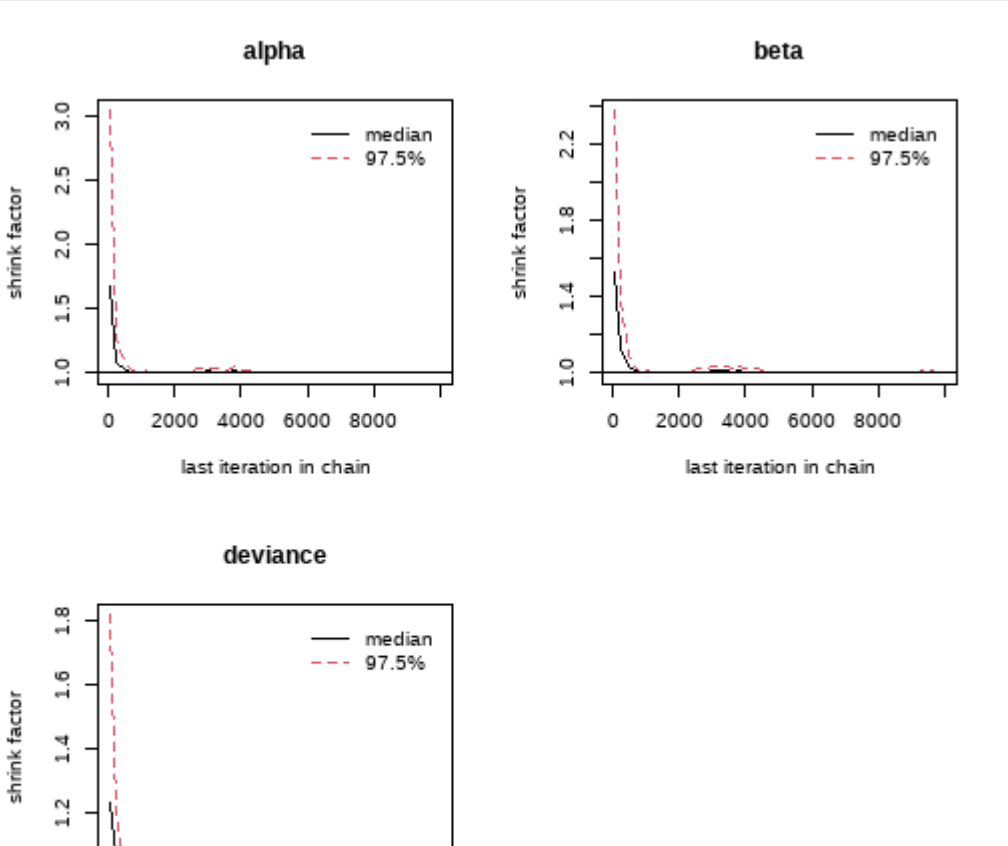
```
gelman.diag(mcmc_model)
```

```
## Potential scale reduction factors:
##
##          Point est. Upper C.I.
## alpha             1       1.01
## beta              1       1.01
## deviance          1       1.01
##
## Multivariate psrf
##
## 1
```

Potential scale reduction factor

We can see how the potential scale reduction factor changes through the iterations using the gelman.plot() function.

```
gelman.plot(mcmc_model)
```



Possible remedies to slow convergence

more iterations, multiple chains

Burn-in

- In theory convergence occurs regardless of the starting point.
- May take a long time to reach high density regions.
- Burn-in is the initial phase of a Markov Chain Monte Carlo (MCMC) simulation where the chain is allowed to converge to the target distribution. Samples from this phase are discarded to avoid bias from the starting values, which may not represent the true posterior.
- We don't know exactly when convergence occurs, so it is not always clear how much burn-in we would need.

Thinning

- Thinning involves selecting every k-th sample from the MCMC chain to reduce autocorrelation and memory usage. This step ensures the retained samples are more independent, though it's not always necessary if autocorrelation is minimal.
- It reduces correlations and increase ESS.

Model checking

An example in reliability analysis

The table hereafter describes the multiple failures of pumps involved in the cooling system of a nuclear power plant. We consider that the number of observed failures y_i of the i-th pump during a time period t_i (in thousands of hours) follow a Poisson distribution $Poi(\lambda_i t_i)$. In order to build a Bayesian model for the number of failures, we specify the following **hierarchical** prior:

$$\begin{aligned} &\lambda_i|\beta \sim \mathcal{G}(\alpha, \beta),\ 1 \leq i \leq 10 \\ &\beta \sim \mathcal{G}(\gamma, \delta) \\ &\perp_{1 \leq i \leq n} \lambda_i|\beta. \end{aligned}$$

with hyperparameters $\alpha = 1.8, \gamma = 0.01\ \delta = 1$.

```
##    pump nb_failures time
## 1     1           5   94
## 2     2           1   16
## 3     3           5   63
## 4     4          14  126
## 5     5           3    5
## 6     6          19   31
## 7     7           1    1
## 8     8           1    1
## 9     9           4    2
## 10   10          22   10
```

Questions

- determine the joint posterior distribution and the full posterior conditional distributions.
- provide 95$\%$ credibility intervals for the λ_i's and for β.
- Imaging that these pumps are connected in parallel, so that the cooling system is working if at least one pump is working. Compute the posterior probability that the system will work well more than 10000 hours (Pumps will not be repaired). *Hint: if* $y_i|\lambda_i, t_i \sim Poi(\lambda_i t_i)$ *then we can show that the time until the first failure is distributed accordingly to an exponential distribution with parameter* λ_i.

Reminder

Reminder: the Poisson distribution for modelling count data (data collected over equal area/time region) is given by

$$y|\theta \sim Poi(\theta)$$
$$p(y = k|\theta) = \frac{\theta^k}{k!}e^{-\theta},\ k = 0, 1, \dots$$

Let us now consider small (time) areas of length $h > 0$. **Assume** that

- if h is small enough, the probability of observing an event on this time interval h is proportional to h

$$P(1 \text{ event within time period of length } h) = \lambda h + o(h).$$

where $o(h)/h \to 0,\ h \to 0$, λ is the intensity of the process.

- the probability of observing more than one event over the time interval h is negligible if h small.

$$P(2 \text{ or more event within } h) = o(h).$$

- events that occur within two disjoint intervals are independent.

Under these conditions,

Jags model

```
library(R2jags, quietly = TRUE)
library(coda, quietly = TRUE)

model_code = '
model
{
  for (i in 1:N)
  {
    lambda[i] ~ dgamma(1.8, beta)
    tau[i] <-  (lambda[i]*t[i])
    y[i] ~ dpois(tau[i])
    # Posterior predictive
    y_rep[i] ~ dpois(lambda[i] * t[i])
  }
beta  ~ dgamma(0.01,1)

}
'
model_data = list(y = c(5, 1, 5, 14, 3, 19, 1, 1, 4, 22),
t = c(94, 16, 63, 126, 5, 31, 1, 1, 2, 10),
N  = 10)
```

Jags model

```
model_parameters =  c("beta","lambda","y_rep")
# Run the model
model_run = jags(data = model_data,
                 parameters.to.save = model_parameters,
                 model.file=textConnection(model_code),
                 n.chains=4,
                 n.iter=10000,
                 n.burnin=200,
                 n.thin=2)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 10
##    Unobserved stochastic nodes: 21
##    Total graph size: 55
##
## Initializing model
```

Posterior check

Posterior summaries

```
print(model_run)
```

```
## Inference for Bugs model at "5", fit using jags,
##  4 chains, each with 10000 iterations (first 200 discarded), n.thin = 2
##  n.sims = 19600 iterations saved
##            mu.vect sd.vect   2.5%    25%    50%    75%  97.5%  Rhat n.eff
## beta         2.399   0.694  1.278  1.904  2.323  2.806  3.965 1.001  7500
## lambda[1]    0.071   0.027  0.028  0.051  0.067  0.087  0.134 1.001 20000
## lambda[2]    0.153   0.092  0.029  0.085  0.134  0.202  0.383 1.001  8200
## lambda[3]    0.103   0.040  0.041  0.074  0.098  0.127  0.197 1.001 20000
## lambda[4]    0.123   0.031  0.070  0.101  0.121  0.142  0.190 1.001 20000
## lambda[5]    0.653   0.305  0.206  0.429  0.605  0.826  1.381 1.001  8200
## lambda[6]    0.622   0.136  0.386  0.526  0.611  0.708  0.918 1.001 13000
## lambda[7]    0.857   0.550  0.156  0.457  0.739  1.131  2.235 1.001 20000
## lambda[8]    0.861   0.555  0.160  0.463  0.745  1.118  2.268 1.001 12000
## lambda[9]    1.358   0.612  0.459  0.914  1.260  1.696  2.807 1.002  3800
## lambda[10]   1.929   0.410  1.215  1.639  1.894  2.186  2.826 1.001 20000
## y_rep[1]     6.617   3.611  1.000  4.000  6.000  9.000 15.000 1.001 20000
## y_rep[2]     2.458   2.165  0.000  1.000  2.000  4.000  8.000 1.001 20000
## y_rep[3]     6.523   3.587  1.000  4.000  6.000  9.000 15.000 1.001 20000
## y_rep[4]    15.550   5.571  6.000 12.000 15.000 19.000 28.000 1.001 20000
## y_rep[5]     3.256   2.360  0.000  2.000  3.000  5.000  9.000 1.001 20000
## y_rep[6]    19.320   6.089  9.000 15.000 19.000 23.000 33.000 1.001  7400
## y_rep[7]     0.858   1.075  0.000  0.000  1.000  1.000  4.000 1.001 20000
## y_rep[8]     0.863   1.094  0.000  0.000  1.000  1.000  4.000 1.001 20000
```

Reliability of the system

Reliability | R code | Posterior density

Let $R_S(t_F)$ be the reliability of a system of pumps in parallel, i.e, the probability that the system is working till the time t_F. Since the time to the first failure of the i-th pump is $t_i|\lambda_i \sim Exp(\lambda_i)$. We readily deduce that

$$R_S(t_F) = 1 - \prod_{i=1}^{10}(1 - \exp(-t_F\lambda_i)).$$

Since we have generated chains from the posterior distribution of the λ's, we can easily get the posterior distribution of $R_S(t_F)$ and therefore we can compute the posterior probability that the system will work more than 10000 hours.

Reliability of the system

Reliability | R code | Posterior density

```
lambda =model_run$BUGSoutput$sims.list$lambda
post_Rf = 1-apply((1-exp(-10*lambda[,1:10 ])), 1, function(x){prod(x)})
mean(post_Rf)
```

```
## [1] 0.8615337
```

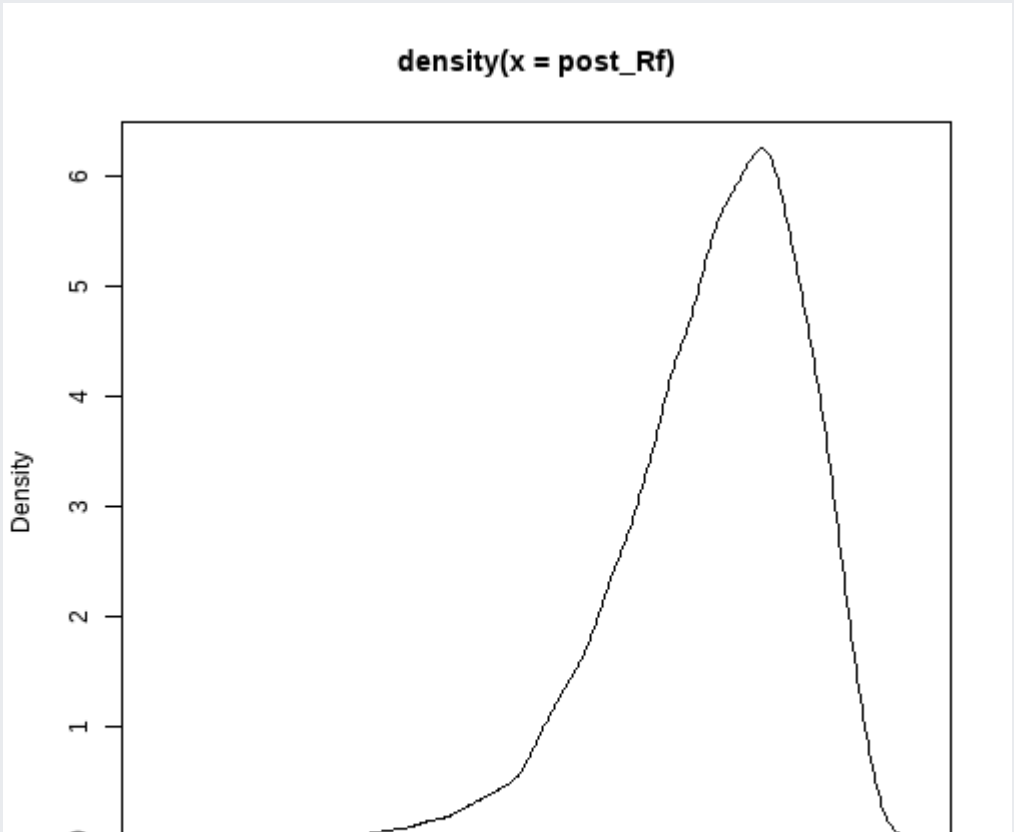
```
quantile(post_Rf, c(0.025,0.975))
```

```
##      2.5%     97.5%
## 0.7030529 0.9662131
```

Reliability of the system

Reliability | R code | Posterior density

```
plot(density(post_Rf))
```



Modeling approach II

$$\lambda_i|\alpha,\beta \sim \mathcal{G}(\alpha,\beta),\ 1 \leq i \leq 10$$
$$\beta \sim \mathcal{G}(\gamma,\delta)$$
$$\alpha \sim \mathcal{U}(0,5)$$
$$\perp_{1\leq i\leq n} \lambda_i|\alpha,\beta.$$

with $\gamma = 0.01$ $\delta = 1$.

Jags model

```
model_code = '
model
{
  for (i in 1:N)
  {
    lambda[i] ~ dgamma(alpha, beta)
    tau[i] <-  (lambda[i]*t[i])
    y[i] ~ dpois(tau[i])
  }
beta  ~ dgamma(0.001,0.001)
alpha  ~ dunif(0.001,5)
}
'
model_data = list(y = c(5, 1, 5, 14, 3, 19, 1, 1, 4, 22),
t = c(94, 16, 63, 126, 5, 31, 1, 1, 2, 10),
N  = 10)
```

Jags model

Algo 1 | Diag 1-alpha | Algo 2 | Diag 2

```
model_parameters =  c("alpha","beta","lambda")
# Run the model
model_run = jags(data = model_data,
                 parameters.to.save = model_parameters,
                 model.file=textConnection(model_code),
                 n.chains=4,
                 n.iter=10000,
                 n.burnin=200,
                 n.thin=1)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 10
##    Unobserved stochastic nodes: 12
##    Total graph size: 45
##
## Initializing model
```

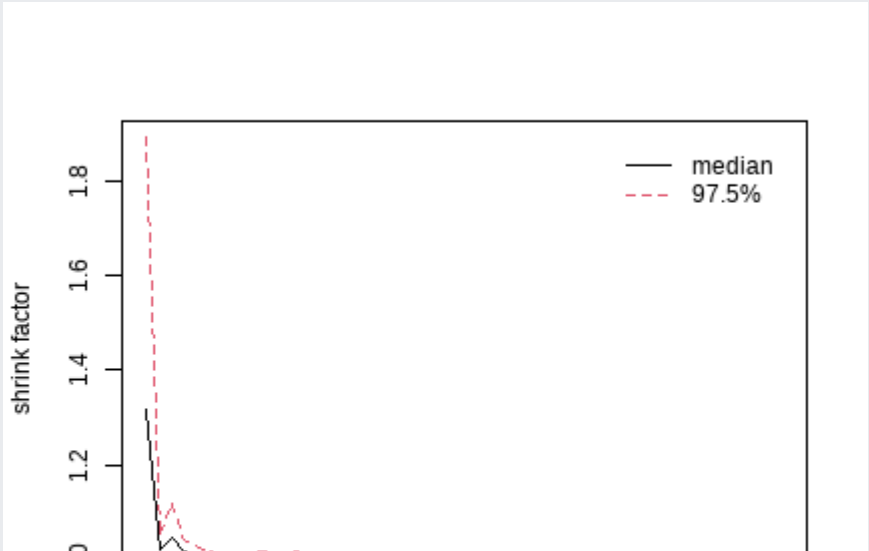
Jags model

Algo 1 | Diag 1-alpha | Algo 2 | Diag 2

```r
# Extract the MCMC samples for the parameters
mcmc_samples <- as.mcmc(model_run)

# Extract only the chains for the "alpha" parameter
alpha_chains <- mcmc_samples[, grep("^alpha", varnames(mcmc_samples))]

# Generate the Gelman plot for "alpha"
gelman.plot(alpha_chains)
```



Jags model

Algo 1 | Diag 1-alpha | **Algo 2** | Diag 2

```
model_parameters =  c("alpha","beta","lambda")
# Run the model
model_run = jags(data = model_data,
                 parameters.to.save = model_parameters,
                 model.file=textConnection(model_code),
                 n.chains=4,
                 n.iter=10000,
                 n.burnin=200,
                 n.thin=4)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 10
##    Unobserved stochastic nodes: 12
##    Total graph size: 45
##
## Initializing model
```

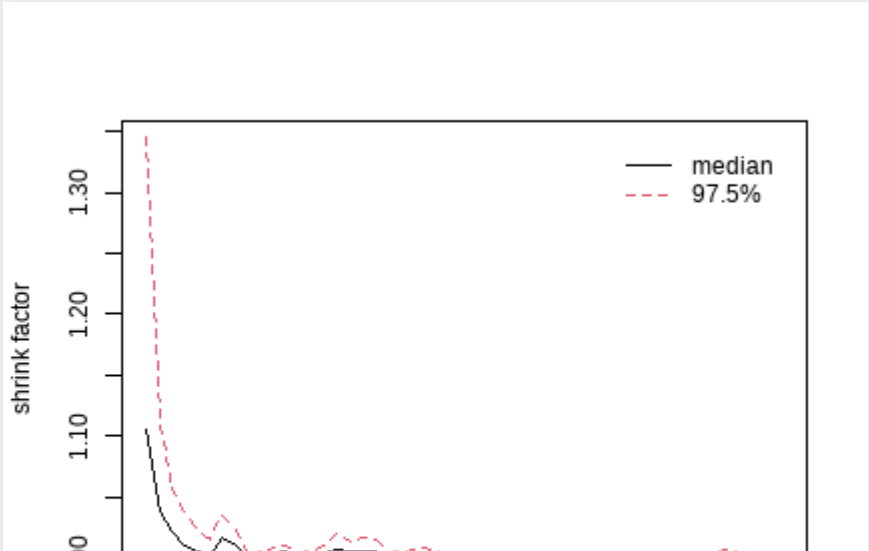
Jags model

Algo 1 | Diag 1-alpha | Algo 2 | Diag 2

```
# Extract the MCMC samples for the parameters
mcmc_samples <- as.mcmc(model_run)

# Extract only the chains for the "alpha" parameter
alpha_chains <- mcmc_samples[, grep("^alpha", varnames(mcmc_samples))]

# Generate the Gelman plot for "alpha"
gelman.plot(alpha_chains)
```



Posterior summary

```
## Inference for Bugs model at "5", fit using jags,
##  4 chains, each with 10000 iterations (first 200 discarded), n.thin = 4
##  n.sims = 9800 iterations saved
##             mu.vect sd.vect   2.5%    25%    50%    75%  97.5%  Rhat n.eff
## alpha         0.946   0.408  0.357  0.651  0.882  1.159  1.932 1.001  9800
## beta          1.502   0.958  0.279  0.817  1.289  1.955  3.948 1.001  9800
## lambda[1]     0.062   0.026  0.023  0.043  0.059  0.077  0.122 1.002  3400
## lambda[2]     0.110   0.082  0.010  0.050  0.091  0.151  0.317 1.001  9800
## lambda[3]     0.092   0.038  0.033  0.064  0.087  0.114  0.181 1.001  8800
## lambda[4]     0.117   0.030  0.065  0.096  0.114  0.135  0.185 1.001  9800
## lambda[5]     0.613   0.319  0.157  0.384  0.558  0.782  1.381 1.001  9800
## lambda[6]     0.613   0.138  0.372  0.512  0.603  0.701  0.915 1.001  9800
## lambda[7]     0.828   0.657  0.088  0.369  0.656  1.101  2.569 1.001  9800
## lambda[8]     0.826   0.650  0.082  0.364  0.661  1.100  2.522 1.001  9800
## lambda[9]     1.468   0.724  0.434  0.942  1.336  1.871  3.189 1.001  6200
## lambda[10]    2.016   0.445  1.248  1.700  1.979  2.291  2.993 1.001  9800
## deviance     43.142   4.413 36.548 39.946 42.426 45.678 53.415 1.002  3100
##
## For each parameter, n.eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor (at convergence, Rhat=1).
##
## DIC info (using the rule, pD = var(deviance)/2)
## pD = 9.7 and DIC = 52.9
## DIC is an estimate of expected predictive error (lower deviance is better).
```

Posterior summary

```
lambda = model_run$BUGSoutput$sims.list$lambda
post_Rf = 1-apply((1-exp(-10*lambda[,1:10 ])), 1, function(x){prod(x)})
mean(post_Rf)
```

```
## [1] 0.9111245
```

```
quantile(post_Rf, c(0.025,0.975))
```

```
##      2.5%     97.5%
## 0.7675457 0.9906081
```